ESQL2 - extending SQL2 to support object-oriented ans deductive databases

Georges Gardarin, Patrick Valduriez

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Rapports de Recherche

1992 25ème anniversaire

N° 1648

Programme 1
Architectures parallèles, Bases de données,
Réseaux et Systèmes distribués

ESQL2 - EXTENDING SQL2 TO SUPPORT OBJECT-ORIENTED AND DEDUCTIVE DATABASES

Georges GARDARIN
Patrick VALDURIEZ

Mars 1992

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

RR-1648

ESQL2 - Une Extension de SQL2 pour gérer les Bases de Données Objets et Déductives

Georges Gardarin[*,**], Patrick Valduriez[*]

[*] INRIA RODIN Project, BP 105, 78153 Le Chesnay

[**] CNRS MASI Laboratory, 45, av. des Etats-Unis, 78000 Versailles

email: georges.gardarin@inria.fr, patrick.valduriez@inria.fr

Résumé

ESQL2 est un langage pour la manipulation de bases de données qui intégre de manière homogène les concepts essentiels des bases de données relationnelles, orienté-objets et déductives. ESQL2 est destiné aux applications traditionnelles aussi bien qu'aux applications avancées comme les grands systèmes d'aides à la décision. En conséquence, les caractéristiques essentielles de ESQL2 sont un système de type riche et extensible, basé sur les types abstraits de données, le support d'objets complexes avec partage référentiel et identité d'objets, la capacité d'interroger et mettre à jour des relations de valeurs ou des classes d'objets en utilisant une syntaxe et une sémantique de type SQL étendu, des fonctionnalités déductives de type DATALOG supportées comme une extension du mécanisme de vue de SQL, et des ordres de contrôle pour programmer des procédures stockées à requêtes multiples. Comme ESQL2 est un langage riche, une représentation graphique des schémas de bases de données et des requêtes ESQL2 est proposée pour simplifier la formulation des requêtes. Une base de données est représentée en utilisant des diagrammes entité-association étendus. Une question est illustrée comme une vue partielle de la base avec des contraintes par prédicats. Cette représentation conduit à un algorithme pour traduire les requêtes ESQL2 en DATALOG étendu avec des identifiants d'objets, des fonctions et des ensembles, ce qui permet de définir la sémantique des questions ESQL2. En résumé, ESQL2 peut être perçu comme une syntaxe compatible SQL pour exprimer des requêtes DATALOG étendu aux objets.

Mots clés

Base de données, langage de requêtes, objet complexe, type abstrait, SQL.

Key-words

Database, query language, complex object, abstract data type, SQL.

ESQL2 - Extending SQL2 to support Object-Oriented and Deductive Databases[1,2]

Georges Gardarin[*,**], Patrick Valduriez[*]

[*] INRIA RODIN Project, BP 105, 78153 Le Chesnay

[**] CNRS MASI Laboratory, 45, av. des Etats-Unis, 78000 Versailles

email: georges.gardarin@inria.fr, patrick.valduriez@inria.fr

Abstract

ESQL2 is an SQL2 upward-compatible database language that integrates in a uniform and clean way the essential concepts of relational, object-oriented and deductive databases. ESQL2 is intented for traditional data processing applications as well as more complex applications such as large expert systems. Therefore, ESQL's salient features are : a rich and extendible type system based on abstract data types (ADTs) which can be implemented in various programming languages ; complex objects with object sharing by combining generic ADTs and object identity ; the capability of querying and updating relations of values or classes of objects using extended SQL syntax and semantics ; a DATALOG-like deductive capability provided as an extension of the SQL view mechanism ; and control statements for programming stored procedures and multi-statement queries. As ESQL2 is a rich language, a graphical representation of ESQL2 database schemas and queries is proposed to simplify query formulation. A database is represented using extended entity-relationship diagrams. A query is pictured as a database view where only relevent parts of diagrams are used ; in addition, types may be constrained by predicates. This representation yields an algorithm to translate ESQL2 queries in DATALOG extended with object identifiers, functions and sets, thereby defining the semantics of ESQL2 queries. Thus, ESQL2 may be perceived as a SQL-compatible syntax for expressing extended DATALOG queries.

---

1 A shorter paper introducing ESQL2 has been published in [Gardarin92] using F-logic semantics. The current paper extends ESQL2 with classes, gives a more comprehensive description of the language based on a graphical representation of databases and queries, and specifies a simpler and clearer semantics using an extended form of Datalog.

2 This work has been funded by ESPRIT project EDS.

1. Introduction

The relational database approach has gained wide acceptance in traditional data processing (i.e., business) mainly because it increases user productivity and enables automatic optimization of database accesses and updates. This approach is highly promoted by the existence of the standard relational query language SQL [ISO89, ISO91, Shaw90] which provides a uniform interface to database administrators, application programmers and end-users for data definition and data manipulation. As a result, SQL is becoming the common language for exchanging data in centralized, decentralized and heterogeneous environments.

Current relational database systems have been designed for traditional data processing applications. Therefore, they do not support well emerging database application domains such as computer aided design (CAD), office information systems (OIS) and knowledge-based systems (KBS), in particular expert systems. These applications express new requirements such as user-defined data types (incorporating both data structures and their associated operations), complex objects (identified values of rich type or complex structure) and rule-based knowledge management. Research aimed at supporting new application requirements in a more integrated way led to two new approaches: object-oriented databases and deductive databases.

Object-oriented databases [Kim90] aim primarily at supporting user-defined data types and complex objects. Their salient features are abstract data types (ADTs) [Guttag77], object identity, type inheritance and persistence independence [Bancilhon88]. ADTs enable the encapsulation of data and their associated operations (called methods) while hiding implementation details. Object identity [Khoshafian86] provides referential object sharing [Khoshafian87] to model complex (graph-structured) objects without replication. Type inheritance enables objects of different types to share the operations pertinent to their common supertypes thereby encouraging reuse of code and reducing the size of application programs. The object-oriented database approach combines object-oriented programming, as exemplified by C++, and database technology. Therefore, database applications can be entirely written in a single database programming language (e.g., as opposed to SQL embedded in a programming language).

Deductive databases focus on integrating within the database the knowledge typically embedded in application programs. With better sharing, control and management of knowledge, typically represented by assertions and deductive rules, deductive databases can

reduce significantly the size and complexity of application programs. The deductive database approach combines logic programming, as exemplified by PROLOG, and database technology. Therefore, it provides a highly expressive database language which allows powerful queries such as recursive queries.

These approaches bring definite advantages over relational databases. However, they generally come with what is perceived by traditional database users as a new language (although it is more a significant extension of an object-oriented or logic programming language). This hampers the acceptance of these technologies by data processing users who could also benefit from their advanced features. Furthermore, albeit it is Turing-complete, a new language does not necessarily solve the infamous "impedance mismatch" problem for the simple reason that there will always be users who want to access the database from their favourite programming language. In other words, the impedance mismatch is not a model or language problem, but must be dealt with by the database system.

A more conservative approach capitalizes on relational database technology and extends it in a way that the advantages of object-oriented databases and deductive databases can be gained. The rationale behind this approach is that the relational model provides a stable platform with simple concepts open for such extensions. The concept of nested relation [Ozsoyoglu87] is useful to model hierarchical structures and manipulate them with a SQL-like language [Schek88]. The concept of domain, not constrained in its original definition, can be defined as ADT [Stonebraker83, Gardarin89a], query or procedure [Stonebraker86], or object [Carey88, Danforth92]. In addition, an extended relational query language can be given a sound functional semantics, e.g., OSQL [Beech88] and FSQL [Valduriez89a], and thus support powerful user-defined methods. Each of these extensions provide some generality over the relational model. However, they have been proposed independently and in various contexts, thereby making their integration difficult.

ESQL2, the language proposed in this paper, is a powerful database language, which integrates relational, object-oriented and deductive constructs, with programming facilities. The ESQL2 data model includes both relational (i.e., value-based) and object-oriented (i.e., reference-based) concepts. The main concepts are those of types, relations, classes, attributes and functions. Types can be elementary types or ADTs. ESQL includes a toolbox for specifying abstract data types. Tuples are instances of relations whereas objects are instances of classes. Both have single-valued or multiple-valued attributes of atomic or complex types. Attribute values can also be references to objects. Functions are associated to types and encapsulate them.

A graphical representation of classes, relations, attributes and functions is proposed to provide a uniform view of ESQL2 database schemas and to simplify database design. The proposed representation, similar to that of DAPLEX [Shipman81] with the introduction of classes are introduced, is an extension of entity-relationship diagrams. Types, relations, classes and attributes are represented by diagrams. Function signatures may also be pictured, according to their argument types, as computed attributes. The graphical representation is also useful to represent queries. More precisely, the semantics of a query is defined in terms of an extended DATALOG rule derived from the graphical representation. Query diagrams are also natural support for query formulation.

Queries are expressed in ESQL2 using the SQL2 framework [ISO91] with additional features to support object-oriented concepts and multi-statement queries. Rich type support is provided by a generic ADT capability allowing multiple implementation programming languages. Complex object support is provided by structures that may contain arrangements of values of any types. Functional expressions are introduced to deal with objects and functions. Also, nesting and unnesting facilities are added to support collections of objects or values.

Futhermore, ESQL2 provides a DATALOG-like deductive capability as an extension of the SQL view mechanism. Recursively derived relations are supported through a view definition facility with controlled levels of recursion. ESQL2 also includes capabilities for programming database procedures using control structures and typed-variables using the database type system. For conciseness, these programming capabilities are not presented in this paper; the reader may refer to [Gardarin92b] for details.

This paper gives a thorough presentation of the most powerful concepts of ESQL2. Section 2 gives an overview motivating the design of ESQL2. Section 3 introduces a powerful ADT model with ADT inheritance and ADT constructors. This model provides a solid basis for supporting efficiently objects of rich type and complex structure. Section 4 illustrates the table and class definition facilities of ESQL2. Section 5 describes the SQL data manipulation extensions for handling complex objects and functions. Section 6 presents the database language constructs introduced for programming multi-statement procedures. Section 7 describes the definition language for integrity constraints. Section 8 presents the ESQL2 deductive capabilities.

2. Model Concepts and Representation

The data model of SQL is an extended relational model, and therefore based primarily on two concepts: domains of values and relations on domains. Furthermore, it supports complex values and complex objects. A complex value is basically a collection of (possibly complex) values with a given structure (e.g., tuple, list, set, array, bag, etc.). A complex object is an identified tuple of (possibly complex) values or objects. Objects are stored in classes while not identified tuples are classically stored in relations. In the following, we introduce the basic concepts of the ESQL2 data model and a graphical representation of schemas.

2.1 Types and Domains

A domain of values in a relation is defined as a data type with possible constraints. The purpose of a domain is to constrain the set of valid values that may be stored in a database. The values of a domain are selected among those of a data type, which may be primitive or complex. A primitive data type is a set of atomic values directly supported by standard SQL2 (i.e., strings, numbers, enumerated types, dates, times, intervals) and comes with operations applicable to values of that type, for instance, arithmetics on numbers. A complex data type is a set of non-atomic values [Ozsoyoglu87], and comes with built-in or user-defined operations. The value of a complex data type instance is always a tuple or a collection of values. We distinguish different sub-types of collections, among them the set, bag, list and array types. A complex data type is viewed as a set of methods (or functions) that operate on values of the defined type. Thus, a complex data type is an ADT; it encapsulates the type structure within a set of operations so that the implementation details are hidden from the user who only sees the interface functions.

2.2 Relations and Classes

A relation (or table) is a bag (or multi-set) of tuples similar to an SQL table. The i-th value in a tuple of a relation is the value of the i-th attribute (or column) in that table. It may be an atomic value or a complex value, according to the type of the i-th column. Tuples do not have identifiers and cannot be referenced directly from other relations, except using key attributes. However, an attribute of a relation can be a reference to an object as shown below.

A class is a set of objects. An object is a tuple of values or objects, with a system identifier. The i-th value in a tuple of a class is the value of the i-th attribute (or column) of that class. Objects in classes are similar to tuples in relations, although they can be referenced from

other relations or classes. In other words, a class is identical to a relation, but contains system identified tuples, called objects.

In a relation or a class, an attribute type can be a class but not a relation. The attribute is then a reference to an object of the class. Class names can also be used as parameters of generic types to construct new types. Thus, a class name defines a new type, which may be used as a complex type. A class is both a type and an extension of that type.

2.3 Functions

Functions (also called methods) are operations encapsulating a given type. The function signature defines the argument and result types, as usual. The function body contains the operation code programmed in low-level language such as C or C++.

An important aspect of functions is that they can be inherited. Generalization and specialization of a type are possible and lead to function inheritance. For example, a complex type is a specialization of a generic type (e.g., a tuple or a set) and as such inherits the generic type built-in functions. Another important aspect of functions is overloading. It is possible to redefine the body of an operation for a subtype. The first argument of a function is used to retrieve the function body since functions encapsulate objects of the type of their first argument. We generally use an object-oriented notation for function applications, where the first argument comes first, followed by a dot, the function name and finally the other arguments, if any, within parentheses. Thus A.F(A1,A2,...An) means that we apply F to A, A1, ...An, F code being attached to A. We also use second order functions, i.e., functions which take functions as parameters; in that case, the function parameter follows the function name in dot notation; for example A.F.f(A1,A2,...An) is a second order function with parameters A, f (a function), A1, A2, ...An.

2.4 Graphical Representation

A graphical representation of classes, relations, types, attributes and functions is proposed to provide a uniform view of database schemas and to simplify database design. The proposed representation is similar to that of DAPLEX [Shipman81], with the addition of classes. Circles indicate types, oblongs indicate relations and rectangles represent classes. A single-headed arrow represents a single-valued attribute, while a double-headed arrow represents a collection-valued attribute. Two types of arrows are introduced to clearly distinguish between traditional attributes and multiple-valued attributes. Type, relation and class names are recorded inside the corresponding circle, oblong or rectangle. Arrows are

labelled by the corresponding attribute names. Function signatures may also be shown on the diagram, as additional arrows going from the source argument to the target argument; intermediate types are necessary for representing functions with multiple arguments; functions can be seen as computed attributes. The proposed graphical representation is defined in Figure 1.

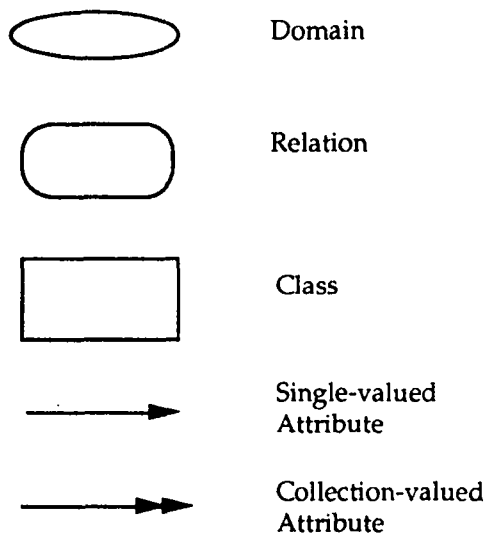

Figure 1: Graphical representation of ESQL concepts

3. DATA TYPES

3.1 Primitive Data Types

The primitive data types of ESQL2 are those of SQL2. A data type is a set of representative values. Each data type includes a null value. SQL2 defines the following data types [ISO91, p67] (for clarity, ESQL2 keywords are in capital letters):

- Character string (CHARACTER, CHARACTER VARYING).
- Bit string (BIT, BIT VARYING).
- Exact numeric (NUMERIC, DECIMAL, INTEGER, SMALLINT).
- Approximate numeric (FLOAT, REAL, DOUBLE PRECISION).
- Datetime (DATE, TIME, TIMESTAMP).
- Time interval (INTERVAL).

3.2 Generic Abstract Data Types

ESQL2 includes a set of generic ADTs to specify and implement ADTs. Generic ADTs are useful for specialization. Generic ADTs are typically parametrized by one or more types. A generic ADT capability is a powerful tool for constructing new complex types; it offers a homogeneous implementation of useful constructors such as tuple and collection. Collections may be specialized into sets, bags, lists or vectors, which are generic classes of most object-oriented database systems.

A simple example is the tuple ADT. It is parametrized with an enumeration of attributes and comes with a set of generic functions such as MakeTuple, Project and Assign. The signatures of the functions acting on a tuple are given in Figure 2. Attribute denotes an attribute name with an associated type. Enumeration of elements is denoted with parentheses. MakeTuple creates a tuple of the generic type from an enumeration of elements. Attributes are the projection functions. Note that there exists one projection function for each attribute, denoted with the attribute name. The equal function assigns the given element to the corresponding attribute of the tuple.

```
ADT Tuple (Attribute, [(,Attribute)...])
(Function to create a tuple from a vector of elements)
        MakeTuple (Element [(,Element)...]) → Tuple
(Functions to project a tuple on a given attribute)
        Tuple.Attribute  → Element
(Function to assign a given attribute of a tuple)
        Tuple.Attribute  = (Element) → Tuple
```

Figure 2: The Tuple generic ADT.

Special attention is paid to the support of collections. Collection is a built-in generic ADT in the language. Collections are organized in an inheritance hierarchy whose root is Collection and subtypes are Bag, Set, List and Vector (see Figure 3).

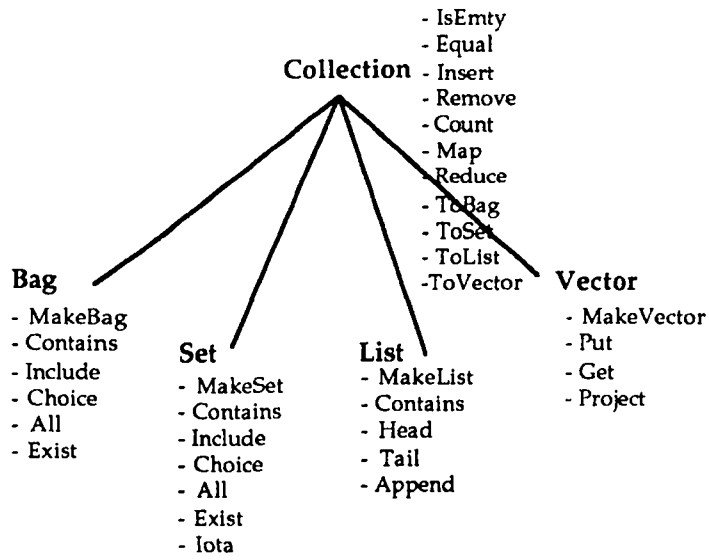

Figure 3: The hierarchy of collections.

The Collection ADT defines any collection of data elements and encapsulates it within the given functions. The signatures of the functions acting on general collections are given in Figure 4. IsEmpty checks if the argument collection is empty. Equal checks if the two arguments are two collections composed of the same elements. Insert inserts a new element in a collection while Remove removes the given element from the given collection. Count counts the number of elements in the collection. Map is a second order function: f(Map(C)) applies the f function to all elements of the C collection and delivers the collection of the results. Map is a powerful function for applying a function to all elements of a collection. Note that the function argument f is written in front, for convenience for query expressions. Reduce is also a second order function which reduces the collection to one element by successive applications of the binary argument function; when the collection is not ordered, Reduce requires a commutative and associative function with a neutral element; the departure value for the function is given as the third argument; for example, if S is a collection of integers, Reduce(S,+, 0) gives an integer which is the sum of the collection of integers. Reduce is, thus, similar to the Pump function of FAD [Danforth92]. Conversion functions are also attached to the collection type. For each subtype of collection x, there exists a conversion function translating a collection y in that subtype: the name of the function is Tox (e.g., ToSet, ToList, ToVector). For example, ToSet(y) transforms the collection y into a set. Other functions are specific to a collection subtype (e.g., set). For each collection subtype, there exists also a Make function, which creates a collection from an enumeration of elements (e.g., MakeSet).

```
ADT Collection (Type):
{Function to check if a collection is empty}
        Collection.Isempty  → Boolean
{Function to check the equality of two collections}
        Collection.Equal(Collection) → Boolean
{Function to insert an element in a collection at the end if ordered}
        Collection.Insert (Element) → Collection
{Function to remove the last inserted element from a collection}
        Collection.Remove (Element) → Collection
{Function to count the number of element in a collection}
        Collection.Count → Integer
{Function to apply a function to all elements of a collection}
        Collection.Map.Function → Collection
{Function to reduce a collection using a pumping function}
        Collection.Reduce (Function, Element) → Element
{Functions to specialize a collection to a specific subtype}
        Collection.ToBag  → Bag
        Collection.ToSet  → Set
        Collection.ToList→ List
        Collection.ToVector  → Vector
```

Figure 4: The Collection generic ADT

All the ADT collection functions are inherited by the subtypes of collection, i.e., bag, set, list and vector. A good example of a generic ADT subtype of collection is the set ADT, as defined in Figure 5. Note that all the collection functions, up to ToVector, apply to sets. MakeSet is the creation function. The other functions respectively check whether a given set contains a given element, whether an element is in a set, whether a set is included in another set. The All function checks whether all the elements of a set satisfy a given formula. The Exist function returns True if at least one element in the set satisfies the given formula, and False otherwise. The Choice function reduces a set by withdrawing an element at random, if any. The Iota function returns the unique element of a set if it is a singleton and empty (null) otherwise. Other functions can be added (e.g., union, intersection and difference of sets). The set generic ADT can be used to define, for example, a set of integers, a set of tuples or a set of complex objects. All these functions can be invoked in ESQL queries as shown in the rest of this paper. Similarly, the bag, list and vector generic ADTs are offered to ESQL users, through the encapsulating functions given in Figure 3.

```
ADT Set (Type) ISA Collection:
{Function to create a set from an enumeration of elements}
        MakeSet (Element [{,Element}...]) → Set
{Function to check wether an element is member of a set}
        Set.Contains (Element) → Boolean
{Function to check is the first set is included in the second}
        Set.Include (Set) → Boolean
{Function to check if all elements of a set satisfy a formula}
        Set.All (Formula) → Boolean
{Function to check if there exists an element satisfying a formula}
        Set.Exist (Formula) → Boolean
{Function to extract an element from a set}
        Set.Choice → Element | ø
{Function to return the unique element of a set}
        Set.Iota→ Element | ø
```

Figure 5: The Set generic ADT.

3.3 Data Type Definition Language

The specification of a new type in ESQL2 requires the use of the CREATE TYPE statement. This statement is built from generic types, which are specialized and nested. It introduces a new type by associating a name to a possibly nested type structure. In the case of an ADT that is a specialization of a previously user-defined ADT, the user must provide its supertypes. Additional attributes may be given when the ADT is a specialization of a tuple type. A type is created using a type definition command: a type name is given to the new type and a complex data type definition is attached to it. The syntax of the CREATE TYPE command is given in annex and illustrated in the next section.

A user constructed type is a specialization of a generic type. As such, it inherits all the functions of the generic types. For example, a set of integers inherits all the functions of the set generic data types, which includes the collection built-in functions. This capability of specializing generic types yields a powerful functional language to manipulate nested objects. However, it is not sufficient to express general functions, which should be provided by a general-purpose programming facility. The type definition may include user function specifications. User functions can also be added later (using the CREATE FUNCTION

command). The type definition may also include integrity constraint specifications. Integrity constraints can also be added later (using the CREATE CONSTRAINT command). In any case, the standard functions supplied in the generic ADT toolbox are usable on a complex data type, according to its structure.

3.4 Type Alterations

The ALTER TYPE statement enables one to modify an existing named type, for instance, adding a new attribute to a tuple type. In the case of a complex data type, the user can change existing operations or modify the subtype relationship. It can also drop a function using the DROP FUNCTION statement or delete an existing type using the DROP TYPE statement.

4. Database Definition Facilities and Examples

4.1 Schema Creation

As in SQL2, a table or a class can be created within a schema, which describes all the elements of a given database. If a schema is not specified, then a schema name equal to the user identifier is implicit. The format of the CREATE SCHEMA command is similar to SQL2. However, type definitions may be included in a schema definition to define the new complex data types of the described database.

4.2 Relation and Class Creation

As previously mentioned, ESQL2 generalizes SQL2 for supporting collections of objects called classes and complex values or references as attributes. Classes and relations only differ from the fact that a class has an implicit attribute, that is, the object identifier. We use the generic term of "entity" to designate either a class or a relation.

The CREATE TABLE statement of SQL is generalized to allow the creation of both relations and classes with attributes of possibly complex types. To support referential object sharing, a type in a tuple definition can be replaced by a class name. The general syntax of the CREATE TABLE statement is similar to that of SQL2, except that it is now possible to create relations and classes. Entities can be altered and deleted as in SQL2, using the ALTER TABLE or ALTER CLASS statement.

An entity can be a specialization of a previously defined entity. The sub-entity clause defines the entity which the new table or class inherits from. Note that a class can inherit from a relation and vice versa; however, a relation does not inherit the object identifier from a class and the object identifier of a class is never inherited. It is possible to rename columns of the generic entities using the AS clause. If AS is not specified and if some column in the set of generic entities appears more than once, then only the first column occurrence is included. Thus, if one subrelation or subclass inherits twice the same attribute from two generic entities, the first occurrence is only kept. This is a simple way to avoid clashes.

4.3. The Engineering Database Example

The Engineering Database Example is derived from a benchmark proposed to measure the kind of performance that engineering applications require [Cattell91]. The database is simply composed of two logical records Part and Connection. The Part class describes identified parts, while the connection relation defines connections between parts. The corresponding ESQL2 database diagram is represented in Figure 9. The database creation commands are given in Figure 6.

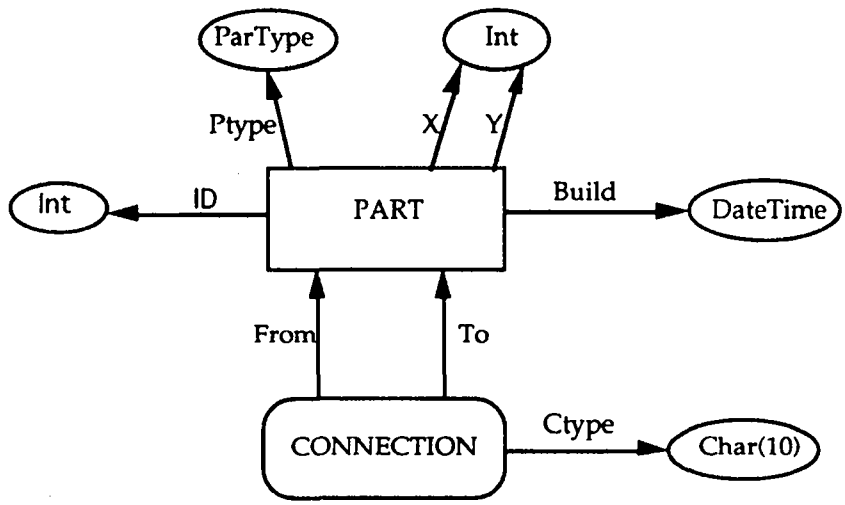

Figure 6: The Engineering Database diagram

```
CREATE SCHEMA EngineeringDb ;
    CREATE CLASS Part (ID Int, Ptype PartType, X Int, Y, Int,
    Build  DateTime) ;
    CREATE RELATION Connection (From Part, To Part,
                Ctype Char(10), Length Int) ;
    CREATE TYPE PartType ENUMERATION (part-type0, part-type1,
          part-type2, part-type3, part-type4, part-type5,
          part-type6, part-type7,  part-type8, part-type9)
          WITH FUNCTION
                LookProc (Ptype PartType, X Int, Y Int),
                InsertProc (Ptype PartType) RETURNS (X Int, Y Int) ;
    CREATE TYPE DateTime TUPLE (DDate Date, TTime Time) ;
```

Figure 7: Creation of the Engineering Database.

5.3 The Portfolio Database Example

Examples of class, table and type creations are given below using a simplified portfolio database. Person is a class whose objects are tuples composed of classical attributes except one of them, which is a tuple (i.e., the person address). Investor is a special kind of Person with an Amount attribute of type money. Share is also a class whose first attribute is the share name and second attribute is a list of tuples describing points in a two dimensional plan. For each share, the final, highest and lowest prices of the last quotation day are given using the money data type. A list of comments describing the major events of the share are given, each comment being well structured as a text. The Sector relation describes the different investment sectors. For each sector, an icon is given as in the Figure data type (i.e., a list of points). The shares belonging to a sector are given in a multi-valued attribute organized as a set. The Dividend relation simply gives the share dividends with their types. The Portfolio class describes the portfolio of the given investors. Investors are objects containing the full description of a person with a type. The Order relation records a relationship between a portfolio and a share and describes share acquisitions. Figure 8 summarizes the design graph of the portfolio database. Complex types are shadowed and not completely defined in terms of elementary types. Further refinements are necessary as shown in the next section. Note that generalization arcs are introduced as dashed arrows (e.g., between Person and Investor). The commands of Figure 9 create the corresponding database schema.

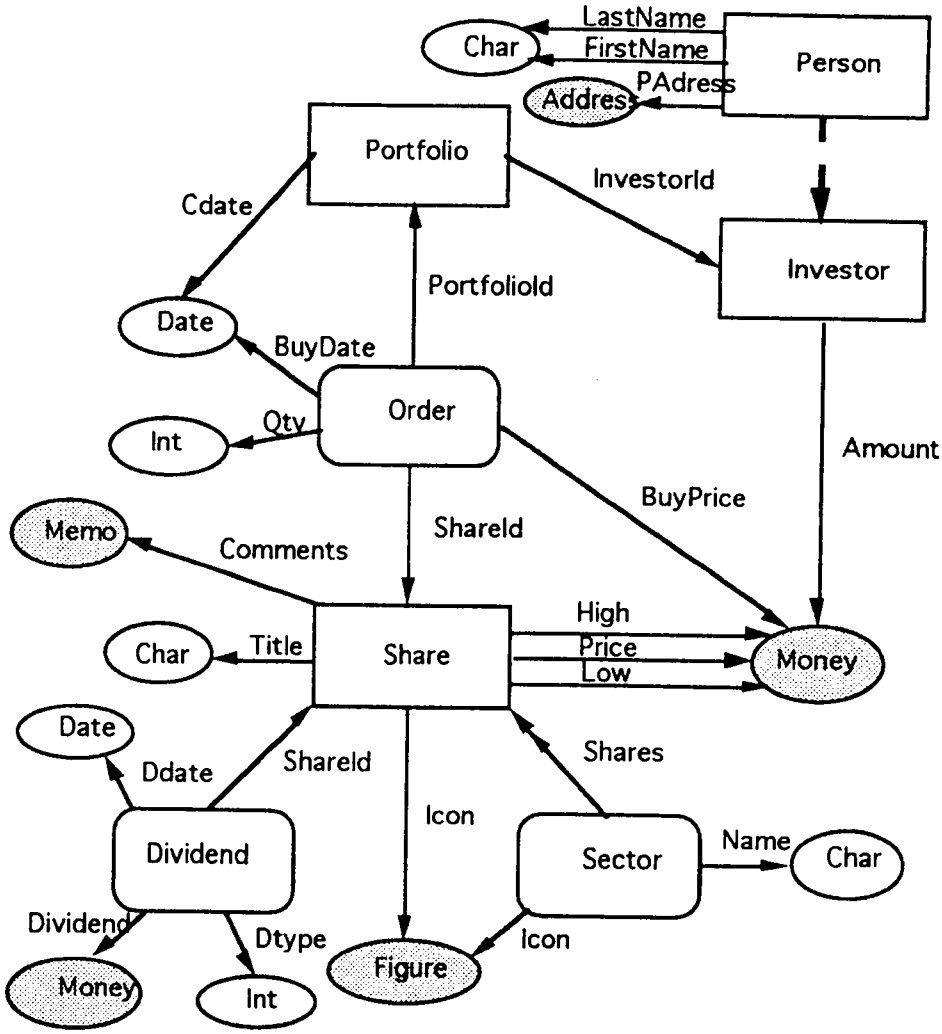

Figure 8: The Portfolio Database diagram.

```
CREATE CLASS Person (LastName Char(20), FirstName Char(20),
    PAddress Address);
CREATE CLASS Investor SUBCLASS OF Person WITH (Amount Money);
CREATE CLASS Share ( Tite Char Varying, Icon Figure, Price Money,
    High Money, Low Money, Comments LIST OF Memo );
CREATE TABLE Sector (Name Char Varying, Icon Figure,
    Shares SET OF Share);
CREATE TABLE Dividend (ShareId Share, Ddate Date, Dtype Int,
    Dividend Money);
CREATE CLASS PORTFOLIO ( InvestorId Investor, Cdate Date);
CREATE TABLE Order (ShareId Share, PortfolioId Portfolio,
    BuyDate Date, Qty Int, BuyPrice Money);

CREATE TYPE Address (Number Int, Street Char Varying, Zip Int,
    City Char Varying, Country Char(20) );
CREATE TYPE Memo ARRAY OF Char Varying
CREATE TYPE Money Numeric(7,2)
CREATE TYPE Figure LIST OF Point
CREATE TYPE Point TUPLE OF (X Real, Y Real)
```

Figure 9: Creation of the Portfolio Database.

5. Semantics of the ESQL Type System

The semantics of a ESQL2 schema is expressed in DATALOG extended with sets and functions [Gardarin89b, Ullman88]. DATALOG predicate and function declarations are derived directly from schema diagrams. To obtain a full semantics, schema diagrams must first be completed. Then, simple rules make it possible to translate a schema in DATALOG declarations. Thus, the ESQL2 type system simply defines DATALOG predicates and functions.

5.1 Completing ESQL Schema Diagrams

As introduced above, it is possible to represent an ESQL2 schema as a graph. Entities (i.e., relations or classes) are pictured as rectangles if they can be referenced (i.e., class), as oblongs if they cannot (i.e., relations). A type or domain (i.e., a constrained type) is represented by a circle. A typed attribute is represented by an arrow pointing from the rectangle or oblong to

the circle. The arrow has a double head when the type is a collection subtype. An attribute that is a class reference is also represented by an arrow pointing to the class.

Methods can also be represented by a graph. Let M(A1 T1, A2 T2, ... An Tn) → (A T) be the signature of a method, where Ai's are argument names and Ti's are type names. A method like M can be represented on a database diagram as in Figure 10. The intermediate transition M is created to keep simple arcs in the graph.

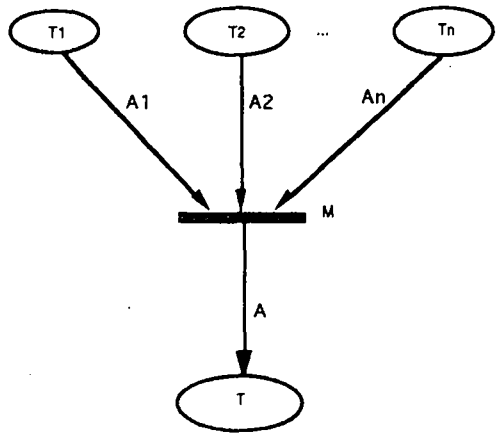

Figure 10: Representation of a method with multiple arguments.

The definition of a specific collection type, say Tc (e.g., a set of integer), as a collection of elements of type T, defines a new type that inherits from the collection generic type c. Thus, all methods that apply to the collection type c become applicable to the specific collection type Tc. The detailed representation of a collection should then show all applicable methods through inheritance on the graph. Thus, a detailed view of the double arrow going from type T to Tc is given in Figure 11. Such detailed representations are not necessary as double arrows always mean collections.

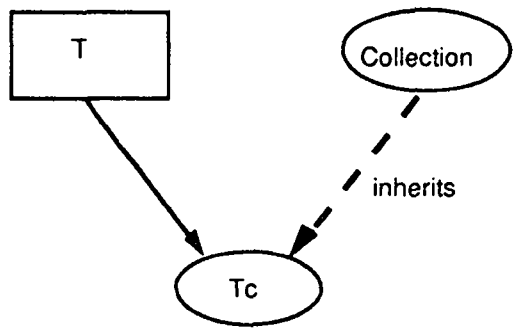

Figure 11: Detailing a Collection.

5.2 Schema Creations as Predicate and Function Definitions

A database schema is a data structure that can be interpreted as a set of predicate and function declarations. A relation is simply translated into a DATALOG predicate whose arguments are the relation attributes. A class is translated in a predicate whose first attribute is the object identifier denoted C* for a class C. Functions can be used to represent both stored mappings from types to types and calculated mappings defined by method codes. Thus, the function, type, class and relation definition statements can be interpreted as DATALOG with functions definition statements.

The translation of an ESQL2 database schema in DATALOG predicate and function definitions can be derived from the graphical representation. Each box in the representation gives a predicate. Each arc joining two types in the representation indicates a function whose domain is the source(s) and codomain is the target. A method with multiple arguments corresponding to a transition is directly represented as a function. Furthermore, methods inherited from generic types are added for completeness, as shown in the previous section. Figure 12 gives the semantics of a significant part of the portfolio database as defined in Figure 8. Function application is denoted using the dot notation as usual.

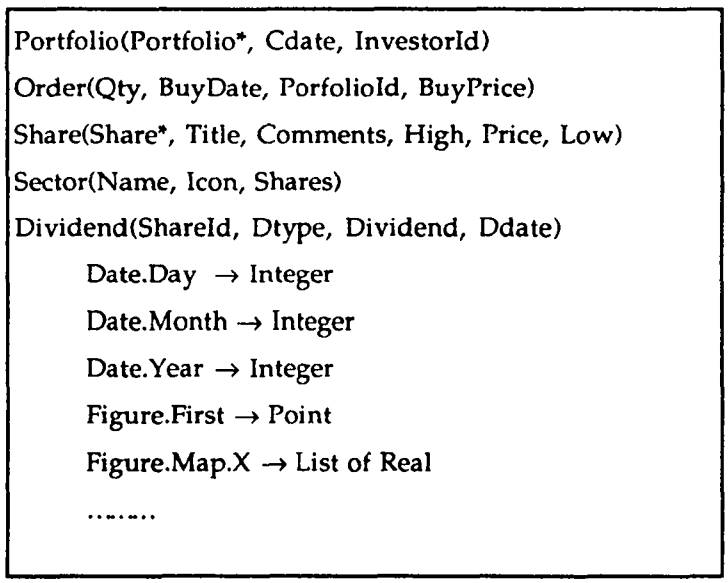

Figure 12: The porfolio database as a set of predicates and functions.

5.3 The Semantics of Inheritance

A relation or a class can be defined as being a sub-entity of an existing entity. Being a specialization of an entity has two effects. First, the sub-entity inherits all attributes of the entity, except the object identifier in case of a relation inheriting from a class. Second, all instances of the sub-entity are instances of the sub-entity. These two properties are simply modelled in DATALOG with a rule of the form:

$E(A_{i1}\ x_{i1}, \ldots, A_{ip}\ x_{ip}) \leftarrow e(A_1\ x_1, \ldots, A_n\ x_n)$

where e is a sub-entity of E. For illustration, the DATALOG rule of Figure 13 gives the semantics of the declaration specifying that an Investor is a person. Note that we use a version of DATALOG in which variables are located by the attribute names, and not by position as usual; this simplifies formulas in general.

```
Person(LastName x, FirstName y, Paddress z) ←
    Investor(LastName x, FirstName y, Paddress z, Amount u)
```

Figure 13: The semantics of Investor SUBCLASS OF Person

Types can also be sub-types of previously defined types. The semantics of type inheritance in ESQL2 is simple and static; it means that functions defined on types are also defined on subtypes. To avoid confusion between function names, the name of a function in the ESQL2 to DATALOG translation should always include the names of the domain and co-domain. For simplicity, we will ignore them when confusion is impossible.

6. QUERY AND UPDATE EXPRESSIONS

In this section, we introduce the language informally through examples. To clarify and prepare the definition of the semantics of queries using extended DATALOG, we illustrate them with query diagrams.

6.1 Representation of Queries

We represent ESQL by diagrams similar to schema diagrams. More precisely, a query is represented by a subgraph of the database graph, possibly with additional arcs and nodes. Classes can be represented as types, since they have the dual property of being both an extension and a type. Each type name is interpreted as a variable on the type. Unary

conditions are introduced to constrain variables within graph nodes, using classical comparison predicates (=, <, ≤, >, ≤, ⊇, ⊃, ∈, ∉, ≠). Constants may be directly written in graph nodes, meaning that the variable must be equal to the constant. Query results are distinguished with a question mark in front of type names. Intermediate result types can also be introduced to represent specific functions such as remove duplicate (a relation type is then introduced) or nest (a grouping function is necessary). Condition boxes can be used to compare different variables. A condition box is a skeleton containing a comparison predicate between variables; condition boxes are linked by an undirected edge to the variables referred by the predicate, from left to right. The following sections detail the query language and the representation diagrams through examples.

6.2 Functional Value Expression

The ESQL2 language is an upward-compatible version of SQL2. The user sees relations or classes and can manipulate them using SQL2. However, since relations or classes may be defined over complex domains, it is also possible to manipulate data inside a complex structure using the ADT functions. Thus, the first significant extension is the possibility of ADT operations. Complex structure instances are encapsulated by their functions as specified in the type definition. Any function attached to a given type may be applied to an element of that type. More generally, a functional value expression is a finite sequence of method applications to a standard SQL value expression. Let A be an attribute expression of type T1. Let Fi be a method defined on type Ti (with possible parameters denoted by Pi) as producing type Ti+1 (i.e., F : T --> Ti+1), for i varying from 1 to n. We use the dot notation to denote function application;. Thus, A.F1(P1) denotes the application of function F1 with parameters P1 to attribute A . A.F1(P1) may be used in any expression of type T2. The general form of a functional attribute expression is A.F1(P1).F2(P2)....Fn(Pn). Thus, SQL expressions are generalized to include the possibility of applying ADT methods at various levels. Expressions and sub-expressions must be correctly typed. For protection, methods that update objects or values cannot be used in queries (SELECT statement) but only in updates (see UPDATE statement).

A value expression can be illustrated by a graph which displays the transformations applied to basic data types. For instance, PAddress.Street and Icon.First.X are valid value expressions, whose diagrammatic interpretations are given in Figure 14.

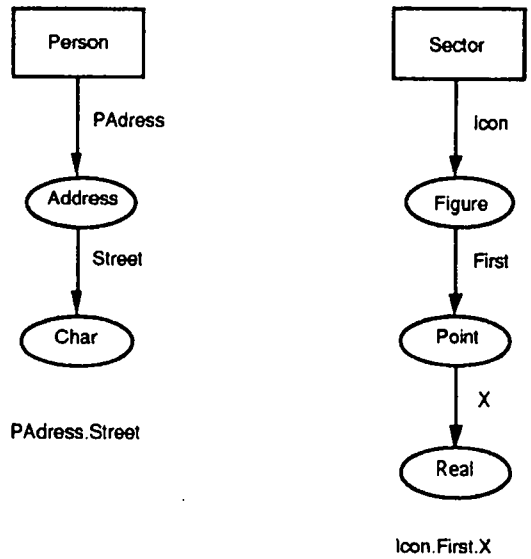

Figure 14: Examples of complex data type value expressions.

6.3 Selection Expressions

A selection takes each tuple from the referenced relation or class, applies a search condition and keeps the selected attributes. Thus, the syntax of the selection clause is similar to that of SQL2, except that classes can now be referenced within the FROM clause. The search condition is defined as in SQL2, including functional value expressions as normal value expressions. Object identifiers are manipulated as special attributes; if Class is a class name, we denote Class* the object identifier attribute.

ESQL2 data manipulation using user-defined or generic functions is now illustrated. For example, let FRANCS be a user function converting MONEY (expressed in dollars) into French francs expressed in the real data type. Query Q1 on the SHARE class lists all the share identifiers for which the price is less than 1,000 francs.

```
(Q1)    SELECT  Share*
        FROM Share
        WHERE  Price.Francs < 1,000
```

This selection expression is illustrated in Figure 15.

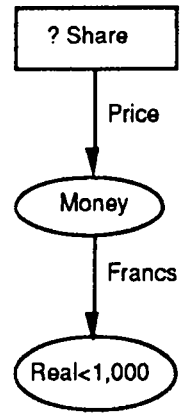

Figure 15: A selection query and its graphical form.

Built-in functions can be used to manipulate collections of objects, which are specific generic ADTs instances. Functions can be composed in expressions. Query Q2 retrieves the name with the icon, the price and the first element of the MEMO array for all the shares of which price is less than 1,000 dollars. We assume here that a DRAW function has been defined on the ICON complex data type, producing a SCREEN data type.

(Q2) SELECT Title, Icon.Draw, Price, Comments.Get(1)
FROM Share
WHERE Price < 1,000

This query is represented in Figure 16.

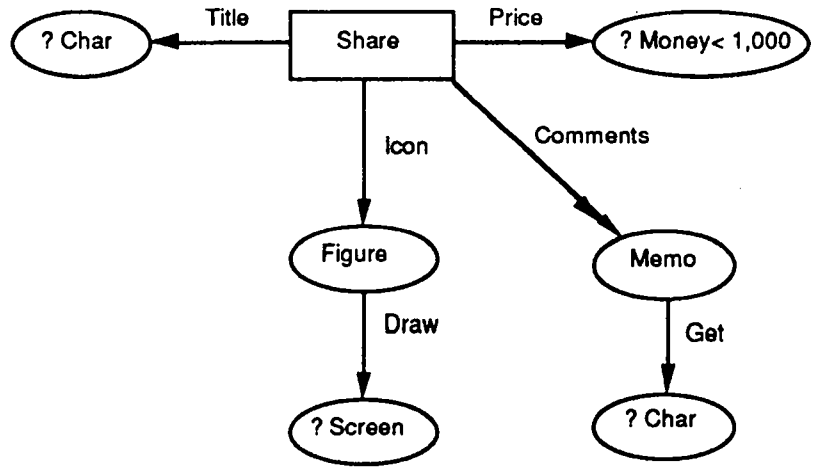

Figure 16: A selection query using built-in and composed functions.

Applying a function to all values of a collection is often desirable. This can be done using the Map function, which is a second order function. Map can be understood as an iterating

function that separates the components of a collection to make it possible the application of functions processing elements. For example, query Q3 retrieves the X coordinates of all points of the Bull's icon.

(Q3) SELECT Icon.Map.X
FROM Share
WHERE Title LIKE "BULL"

This query is represented in Figure 17.

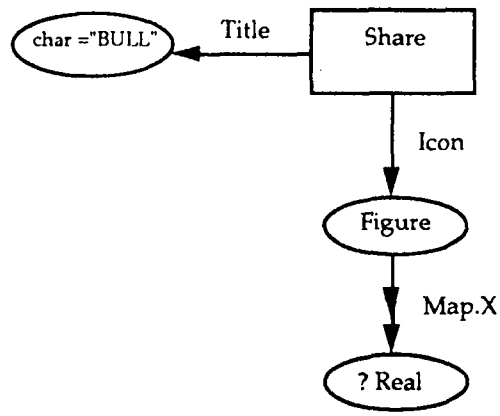

Figure 17: A query using the Map function.

6.4 Value-based Join Expressions

Joins between entities are possible in ESQL2. Value-based joins are performed by comparing the values of attributes of two entities (class or relation) having comparable attribute types. The syntax of value-based join is unchanged from SQL2, functional value expression being now possible. To represent join by values and different form of joins in query diagrams, special join predicates must be displayed in condition boxes. Query Q4, represented in Figure 18, illustrates a natural join between Portfolio and Order based on the equality of dates.

(Q4) SELECT O.Qty, P.Porfolio*
FROM Order O, Portfolio P
WHERE O.Date = P.Date

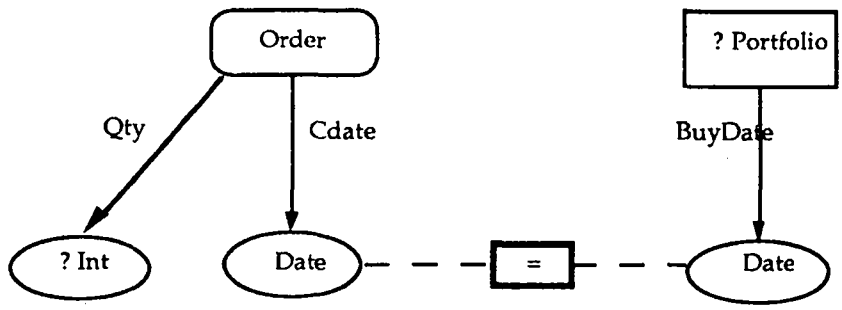

Figure 18: A query performing join on equality of values.

Query Q5, pictured in Figure 19, illustrates an inequi-join between Share and Order using the condition "BuyPrice less than Price".

(Q5) SELECT O.Qty, S.Share*
FROM Order O, Share S
WHERE O.BuyPrice < S.Price

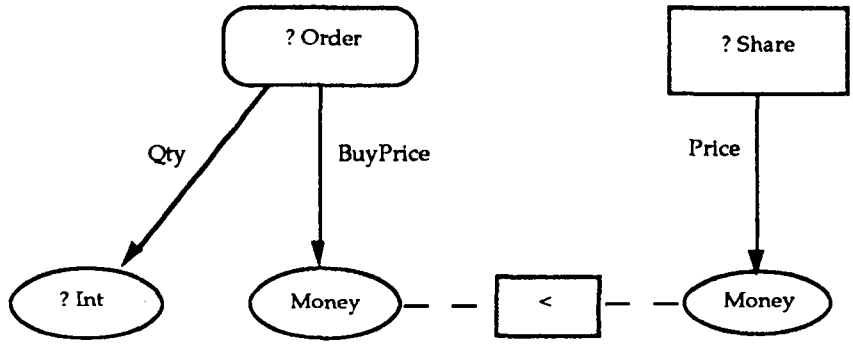

Figure 19: A query performing a theta-Join.

6.5 Reference-based Join Expressions

Join through references is possible in ESQL2 in two ways. First, the reference attribute can be used as a function in the WHERE condition. Second, functional value expressions can be used in SELECT clauses to select attributes reached from a join. Queries Q6 and Q7 illustrates these possibilities. Both queries retrieve the current price of the shares in orders.

(Q6) SELECT Price
FROM Order O, Share S
WHERE O.ShareId = S

(Q7) SELECT ShareId.Price
FROM Order

In query Q6, Share is considered as a class extension while, in query Q7, it is considered as a type. This is possible because of the dual aspects of a class, which is both a type and a container of instances. Thus, the query graphs corresponding to queries Q6 and Q7 are slightly different views of the database diagram, as shown in Figure 20.

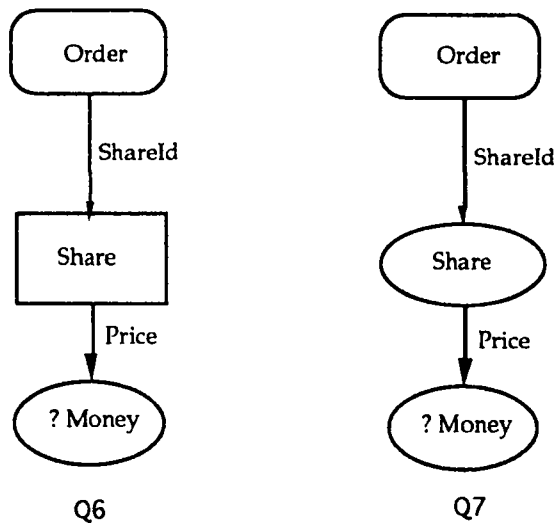

Figure 20: Queries performing joins through references.

6.6 Joining Through Collections

Built-in functions can be used to manipulate collections of objects, which are specific generic ADTs instances. This is helpful for joining through multi-valued references. For example, query Q8 illustrated in Figure 21 retrieves the name with the icon and price of all the shares that belong to the computer sector. It performs a join using the set membership function CONTAINS, applied to a set of Share objects.

(Q8) SELECT S.Title, S.Icon.Draw, S.Price
FROM Share S, Sector R
WHERE R.Shares.CONTAINS(S)

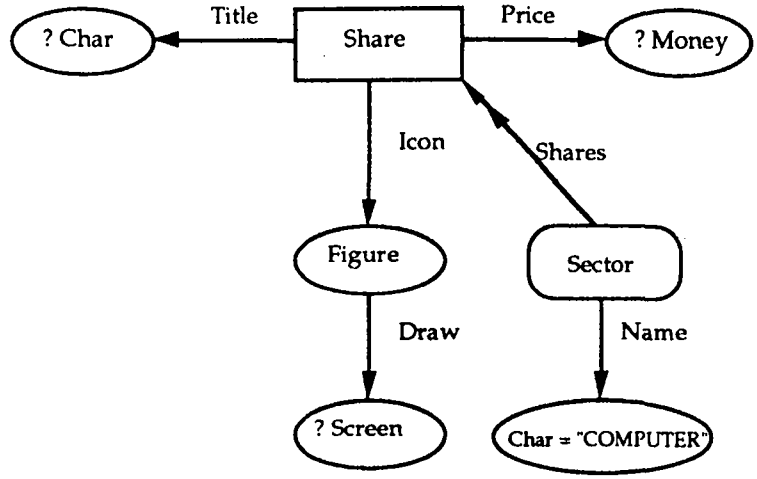

Figure 21: *A query performing a join through multi-valued references.*

When a double-headed arrow is crossed, a collection is obtained as a result of the corresponding function. The functions defined on collections can then be applied, including the Map function. For example, query Q9 retrieves the name and icon of the sector which BULL belongs to. Note that an attribute function (i.e., Title) is applied to a set of tuples (Shares) having that attribute using an intermediate MAP function. Such an application returns the set of attribute values of the tuples. CONTAINS is the Boolean function defined on sets to determine whether an element (here "BULL") belongs to a set.

```
(Q9)    SELECT Name, Icon
        FROM Sector
        WHERE Shares.MAP.Title.CONTAINS("BULL")
```

An alternative formulation of that query is possible without MAP as follows:

```
(Q10)   SELECT Name, Icon
        FROM Sector, Share S
        WHERE Shares.CONTAINS(S) AND S.Title = "BULL"
```

Both queries have a similar representation, except that Share is a type in Q10. The representation of Q10 is given in Figure 22.

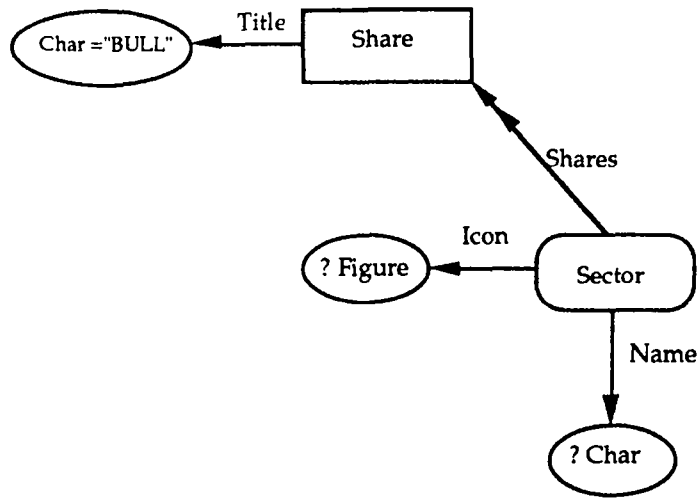

Figure 22: A query performing a join through collections with nested functions.

As an example of set manipulation, query Q11 illustrated in Figure 23 draws the icon of the sector which BULL, SIEMENS and ICL belong to. Note that Share is considered as a collection type in that query, which requires a MAP function to project each share of the collection on Title. The query qualification uses two set functions (Map and Include) and a complex constant, which is a set of three elements (BULL, SIEMENS, IBM).

```
(Q11)   SELECT Icon.Draw
        FROM Sector
        WHERE Shares.MAP.Title.INCLUDE({"BULL","SIEMENS","IBM"})
```

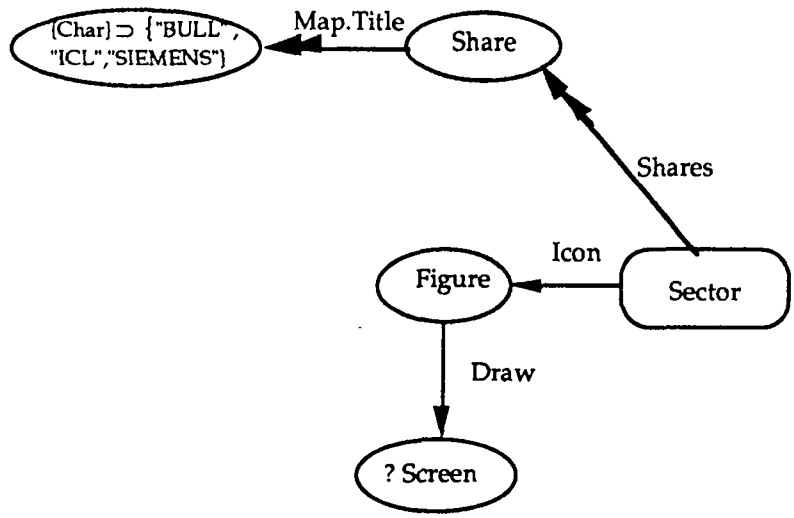

Figure 23: A query performing a join through collections with set comparisons.

6.7 Grouping and Ungrouping Multi-valued Attributes

Collection handling requires the ability to transform single-valued attributes into collections according to a grouping criterion and a result schema. This is the classical "nest" operation, generally implemented by a GROUP BY in SQL. In ESQL2, we maintain the GROUP BY clause. However, the result of a GROUP BY does not require applying an aggregate function (e.g., SUM): it can simply be a collection constructing function. Thus, we use the function translating a collection into the desired type to specify the results (e.g., TOSET). Thus, query Q12 gives a set of non expensive shares (price less than 1,000) for each sector. This query is illustrated in Figure 24. Note that a second order Group function is introduced with the grouping attributes as parameters. This is necessary to indicate the grouping attribute.

(Q12) SELECT Name, S.Title.TOSET
FROM Share S, Sector
WHERE Price < 1,000 AND Shares.CONTAINS(S)
GROUP BY Name

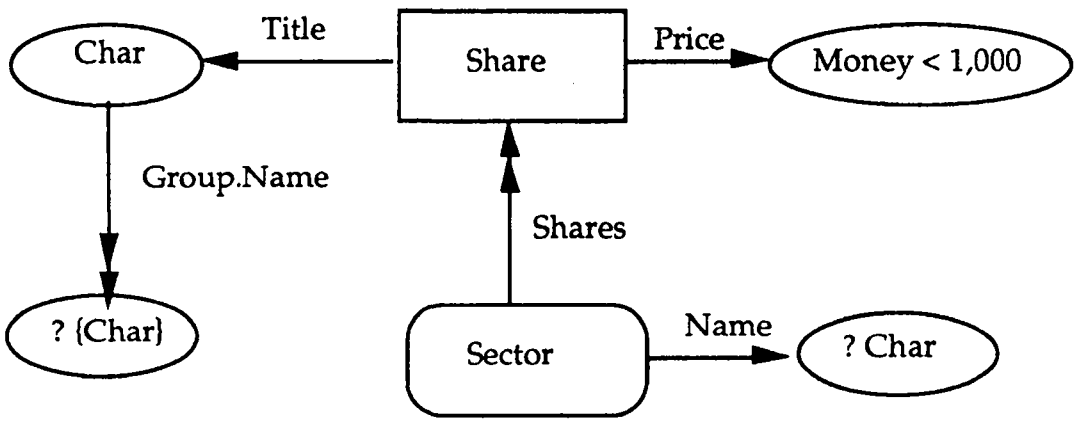

Figure 24: A query performing a grouping.

To get another type of collection (e.g., a bag), the selected collection type transformation function (e.g., TOBAG) must be applied. It is also possible to use several functions constructing collections in the projection expression. Functions can be applied to collection valued attributes. Thus, query Q13 represented in Figure 25 first computes the set of dividends per type, for the dividends paid after the 1st of January 1990. Next, it applies the Reduce function to reduce each set of dividends to its sum. A set being without duplicate, the result is the sum without duplicates. The standard SUM(MONEY) aggregate function of

SQL would give a slightly different result, i.e., the result would be the sum with duplicates. Note that the aggregate function SUM(MONEY) is equivalent to reducing a list using addition, which may be done using the target MONEY.TOLIST.REDUCE(+,0). The query also gives a list of dates of payment for each dividend type.

(Q13) SELECT Dtype, Dividend.TOSET.REDUCE(+,0), Ddate.TOLIST
FROM Dividend
WHERE Ddate > 01-01-90
GROUP BY Dtype

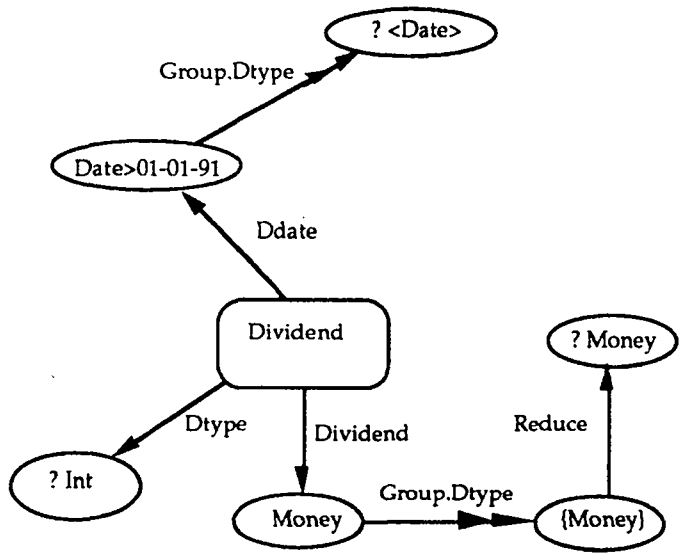

Figure 25: A query performing a grouping and a reduction.

It is also possible to unnest a relation with a multi-valued attribute. The specification of the unnested attribute(s) is done using a flattening function denoted FLATTEN. In the case of multiple flattening functions appearing in the query target list, a full unnesting is performed. For each value of the ungrouped attribute list, tuples are composed with each element of the Cartesian product of the corresponding values of the other attributes. For example, query Q14 illustrated in Figure 26 gives, for each share title, the corresponding sector name, that is, it flattens the SECTOR relation with the depending share titles.

(Q14) SELECT Shares.MAP.Title.FLATTEN, Name
FROM Sector

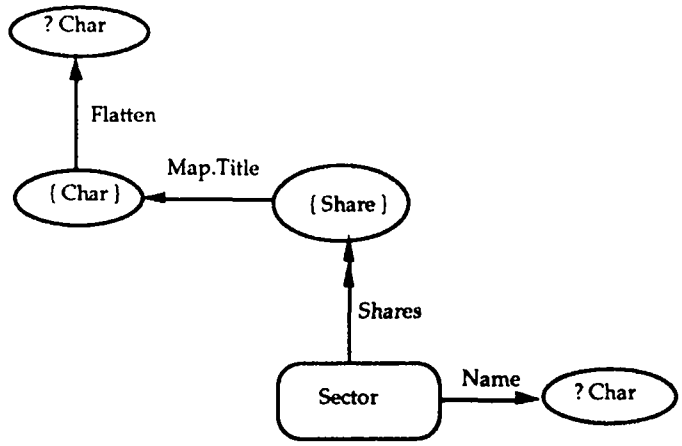

Figure 26: A query performing a flattening.

6.8 Updating Relations and Classes

The UPDATE command of SQL can now be used to modify values or references in relations or classes. First, objects must be inserted in classes using a classical INSERT statement, as done for example in update U1, where Tree is a list of points. Object identifiers are generated by the system.

```
(U1)    INSERT INTO Share(Title, High, Price, Low, Icon)
        VALUES("BULL", 50, 47, 32, Tree)
```

Next, objects are addressed through sub-queries that retrieve the object identifier in some relation or class. A function that updates a complex object can be called from a query. For security reasons, this is forbidden: the key word UPDATE should be used for calling updating functions. Collections are updated through methods. For example, one can add a new share in the computer sector using the Insert function for collections, as done in (U2). The "=" of SQL after SET is replaced by "." as the command does not assign a new value to a complex attribute (here Shares), but rather applies a function to it; "=" will still be used in case of assignment.

```
(U2)    UPDATE Sector
        SET Shares.INSERT (Share*)
        WHERE Name = 'COMPUTER'
        AND Share* =
                SELECT Share*
                FROM Share
                WHERE Title = "BULL"
```

Inserting a complex object in the database requires object creation. A complex object or value can be created using the Make function (MakeTuple, MakeBag, MakeSet, MakeList, MakeVector) or any nesting of them. We may use also the external notations with parentheses for denoting complex objects or values, i.e., [] for tuple, | | for bag, {} for set and <> for list.

7. Integrity Constraints

ESQL2 supports integrity constraints similar to SQL2. This includes type and entity integrity constraints. Type integrity constraints extend the domain constraints of SQL. They support integrity constraints involving complex values and objects through functions. Relation integrity constraints are naturally extended to complex data by means of functions embedded in search qualifications. Integrity constraints are defined either at relation or class creation, or using a CREATE CONSTRAINT statement. An integrity constraint can be deleted using a DROP CONSTRAINT statement.

7.1 Type Constraints

A type constraint is created using a CREATE CONSTRAINT command applied to a data type. The type constraint specializes the data type by restricting the admissible values using a search condition. The search condition of a domain constraint must only contain references to functions defined on the data type. In particular, it must not contain column specifications, which must be replaced by the data type name. For example, the statement I1 creates an integrity constraint, which states that all French zip codes must be among 01000 and 95999:

```
(I1)    CREATE CONSTRAINT FrenchZip ON TYPE  Person
        CHECK (NOT PAdress.Country = 'FRANCE')
        OR (PAdress.Zip BETWEEN 01000 AND 95999)
```

7.2 Entity Constraints

An entity constraint specifies an integrity constraint that involves one or more relations or classes. As in SQL2, there are three kinds of relation constraints: unique constraint, referential constraint and check constraint. A check constraint specifies a condition for a relation.

We now give a few table constraint examples using the PORTFOLIO database. The first one (I2) simply specifies that name is the primary key in table SECTOR. The second one (I3) specifies that SHAREID in table DIVIDEND is a foreign key in class SHARE. Note that referential integrity is now posible from a relation to a class. The third one (I4) states that a share can be ordered only after a portfolio has been created for the corresponding investor.

(I2) CREATE CONSTRAINT KeySector ON TABLE Sector
PRIMARY KEY Name

(I3) CREATE CONSTRAINT RefShare ON TABLE Dividend
FOREIGN KEY (ShareId) REFERENCES Share
ON DELETE CASCADE

(I4) CREATE CONSTRAINT Precedence ON TABLE Order
CHECK BuyDate ≥ PortfolioID.Cdate

8. Views and Deductive Facilities

Much work has been devoted to extending relational databases to deductive databases. The DATALOG language allows the user to derive virtual relations from the database. DATALOG has the power of relational algebra plus recursion. It may be extended to support negation, functions and sets. Unfortunately, the formal syntax of DATALOG does not fit well with that of SQL [Gardarin89b]. Given their importance, recursive queries have been introduced in SQL3 with a rather complex syntax.

ESQL2 fully supports DATALOG queries with an SQL like syntax. Deductive capabilities are introduced through derived relations defined as generalized views. We simply extend the view concept of SQL with query expressions, including unions and differences of SQL SELECT statements. We also support recursion, by allowing a view to be defined from itself. Deriving a recursive relation from base relations is a powerful capability. As an example, consider the table CONNECTION (From Part, To Part) of the Engineering Database Benchmark. The recursive view definition V1 specifies the transitive closure of the CONNECTION relation:

```
(V1)    CREATE VIEW Closure (From, To) AS
            SELECT From, To
            FROM Connection
        UNION
            SELECT O.From, L.To
            FROM Connection C, Closure L
            WHERE O.To = L.From ;
```

This command can be illustrated by two diagrams (see Figure 27), one for each SELECT, which are built as usual. Recursion is almost invisible, as it comes because the view itself is referenced in the second SELECT.

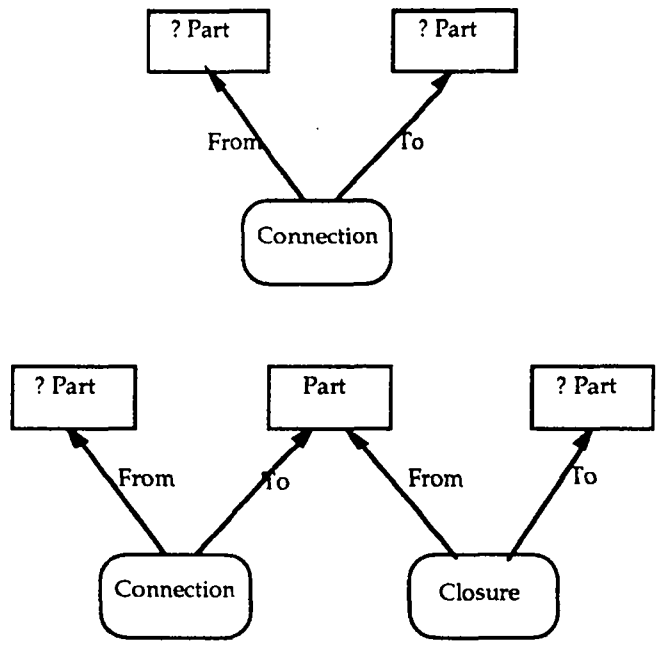

Figure 27: Diagrams of the Closure Recursive Query.

The Cattell Benchmark [Catell91] requires the execution of two interesting procedures:

- Lookup, which generates 1000 random part IDs and fetches the corresponding parts from the database; for each part, call a procedure NullProc(Type, X, Y), with no effect.
- Traversal, which finds all parts connected to a randomly selected part, or to a part connected to it, and so on, up to 7 hops; for each part, call a procedure NullProc(X, Y) attached to Type, with no effect.

We give as query (Q15) and (Q16) the database procedures that could be used to implement the benchmark. Q15 requires a programming language facility that is also part of ESQL2 [Gardarin92b]. In the second example, we introduce a limitation on the number of times the recursive relation is crossed (i.e., the number of times Traversal is used to

construct the view). This "RECURS n TIMES" statement is important to satisfy Cattell's requirements.

```
(Q15)   PROCEDURE Lookup();
        VAR i Int = 0 ;
        VAR Npart = 1000 ;
        WHILE i < 1000 DO
                BEGIN
                i = i+1;
                SELECT Type.NullProc(X, Y)
                FROM Part P
                WHERE P.ID = Random(Nparts) ;
                END{WHILE} ;
        END{Lookup}
```

```
(Q16)   PROCEDURE Traversal();
                CREATE VIEW Traversal AS
                SELECT C.*, Type.NullProc(X, Y)
                FROM  Connection C
                WHERE  C.From.ID = Random(Nparts)
        UNION
                SELECT SELECT C.*, Type.NullProc(X, Y)
                FROM Traversal T, Connection C
                WHERE T.From= C.To
        RECURS 7 TIMES ;
        END{Traversal}
```

Note that ADT functions can be used in recursive queries, thereby giving ESQL2 the power of DATALOG with functions. However, one problem with using ADT functions in recursive queries is that they may lead to infinite sets of answers.

To further illustrate the recursive facility, let us assume a ROADS relation, representing the roads between large cities of EUROPE, created as follows:

```
CREATE TABLE Roads ( R# INT, Scity CHAR(20), Tcity CHAR(20), Length FLOAT ) ;
```

An example of a derived table with functions is a PATHS relation, which gives all paths from one city to another with their lengths. Note that the PATHS relation is virtually infinite, as cycles generally exist in road graphs. Nevertheless, certain queries (e.g., find the minimum length path between PARIS and MANCHESTER) have a finite answer. The PATHS relation is defined as follows:

```
(V2)    CREATE VIEW Paths AS
            SELECT Scity, Tcity, Length
            FROM Roads ,
        UNION
            SELECT P.Scity, R.Tcity, P.Length + R.Length
            FROM Paths P, Road R
            WHERE P.Tcity = R.Scity ;
```

To support negation as in certain versions of DATALOG, we use the EXCEPT command of SQL2. For example, if one wants to avoid all paths going through special roads recorded in a FORBID relation, we can define the following view:

```
(V3)    CREATE VIEW Paths AS

        (   SELECT Scity, Tcity, Length
            FROM Roads
        EXCEPT
            SELECT Scity, Tcity, Length
            FROM Forbid  )

        UNION
            SELECT P.Scity, R.Tcity, P.Length + R.Length
            FROM Paths P, Road R
            WHERE P.Tcity = R.Scity ;
```

As shown in the semantics study of ESQL2, the proposed deductive view definition language, with query expression and recursion, has the power of DATALOG with negation and functions. However, a more formal language, with a truly deductive form (i.e., Horn clauses) and updates (i.e., triggers), could be of interest to program complex queries and to maintain database integrity.

9. The Semantics of ESQL2

As shown above, the semantics of an ESQL2 schema is a set of DATALOG predicate and function declarations. Inheritance relationships between classes or relations are translated into a DATALOG rule. Thus, it is natural to translate any query in DATALOG with object identifiers, collections and functions. ESQL2 may be perceived as a SQL-like formulation of DATALOG with object identifiers, collections and functions. Supersets of this kind of DATALOG language has been widely formalized, for example in IQL [Abiteboul90] or F-Logic [Kifer90]. Collection is an abstraction of the set concept, as generally handled in DATALOG extensions. A full semantics for ESQL requires collections with different types of functions for the collection sub-types, i.e., set, bag, array and list. To clarify rules, we use capital letter variables for variables representing collections, normal letters for variables representing single values and normal letters followed by * for variables representing object identifiers.

9.1 Algorithm to translate ESQL2 queries into DATALOG rules

In this section, we sketch an algorithm to translate an ESQL2 query into one or several DATALOG rules. This algorithm is based on the transformation of query diagrams. When the query is composed of several diagrams implicitly connected by union, it generates several rules, one for each diagram. The following transformations are applied from a given diagram:

(1) Each type node (circle) is labelled by a variable u, v, w, ... if it is not the target of a double headed arrow and by a variable X, Y, Z, ... if it is the target of a double-headed arrow; the variable replaces the type name, which is useless in DATALOG.

(2) Each class box is labelled by an object identifier variable r*, s*, t*, ...

(3) Each class box is translated in a rule body predicate whose arguments are the object identifier variable and the adjacent variables labelled by the attribute name (i.e., the name or the class followed by * or the name on the arc going from the class to the type or class).

(4) Each relation box is translated in a rule body predicate whose arguments are the adjacent variables labelled by the attribute name (i.e., the name on the arc going from the class to the type or class).

(5) Predicates in type nodes or condition boxes are translated in additional rule body conditions. Additional conditions must be added to make equal identical variables having the same meaning and to make a simple variable member of set variable when varying on the same class or type.

(6) Arcs connecting two types are translated into functional equations added to the rule body.

(7) If no view is created (normal SELECT), a rule head predicate having name Answer is generated. If a view is specified, a rule head predicate with the name of the view is specified. In any case, the arguments of the head predicate are the variables corresponding to queried types (variables having a question mark in front).

The translation algorithm is illustrated in Figure 27 with graphical rules for most of the transformations.

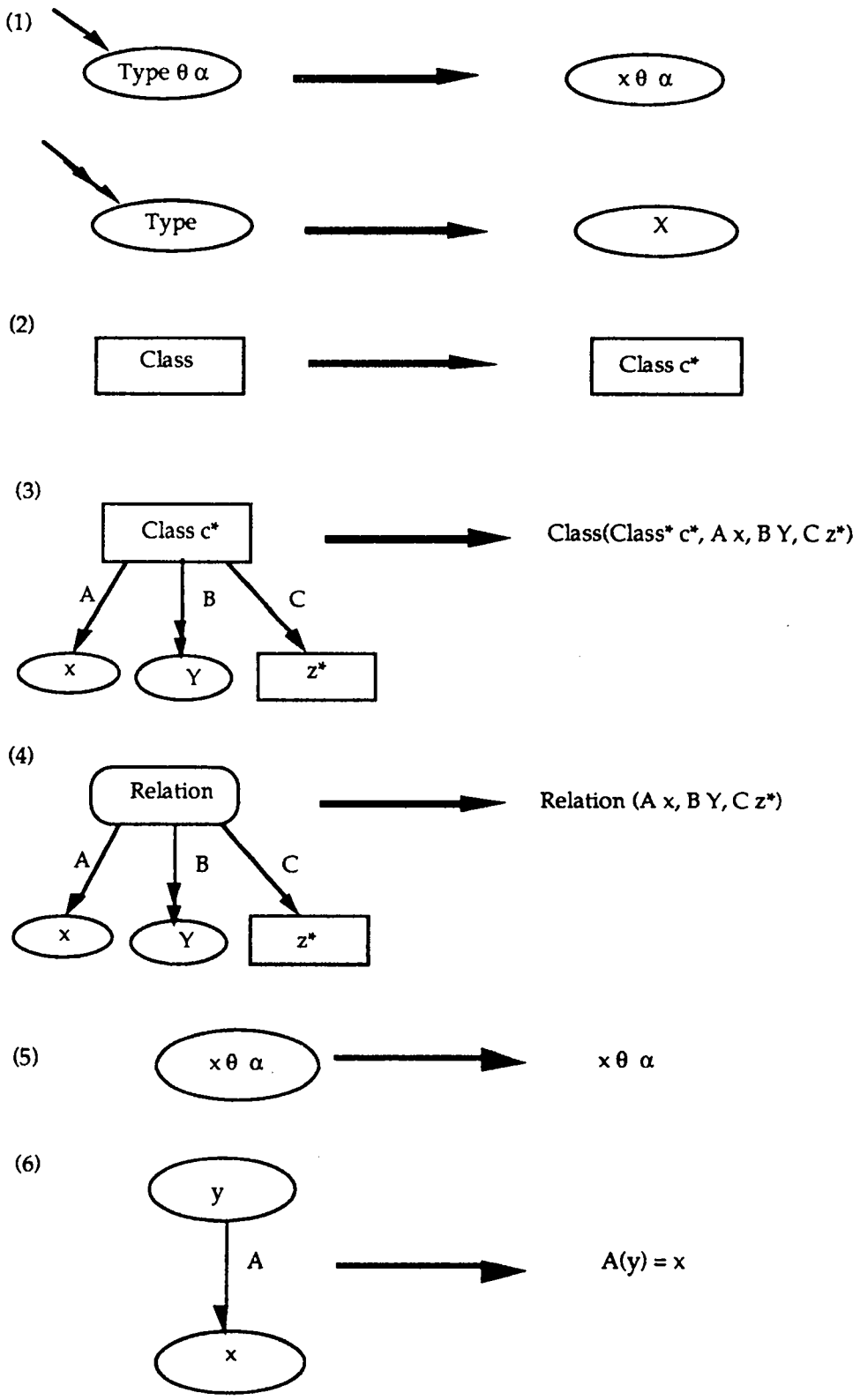

Figure 27: Illustration of rules for translating ESQL2 queries in Extended DATALOG

9.2 Exemples of Translations

In this section, we give the translation in extended DATALOG of all queries represented with a diagram in the above sections. The translation process simply follows the given rules, with a few simple variable manipulations.

(Q1) Answer(Share* s*) ←
Share(Share* s*, Price m), Francs(m) = r, r < 1000

(Q2) Answer(Title t, Char x, Screen y)←
Share(Share* s*, Title t, Price m, Comments C, Icon f),
m < 1000, GET(C,1) = x, Draw(f) = y

(Q3) Answer(SetofReal S)←
Share(Share* s*, Title t, Icon f), MAP(f,X) = S

(Q4) Answer(Qty q, Portfolio* p*)←
Order(Qty q, Cdate d), Portfolio(Portfolio* p*, BuyDate b), d = b

(Q5) Answer(Qty q, Share* s)←
Order(Qty q, BuyPrice b), Share(Share* s*, Price p), b < p

(Q6) Answer(Price p)←
Order(ShareId s*), Share(Share* s*, Price p)

(Q7) Answer(Price p)←
Order(ShareId s), Price(s) = p

(Q8) Answer(Title t, Price p, Draw x)←
Share(share* s*, Title t, Price p, Icon f), Sector(Shares S, Name n),
n = "COMPUTER", Draw(f) = x, s* ∈ S

(Q10) Answer(Name n, Icon f)←
Share(Share* s*, Title t), Sector(Shares S, Icon f, Name n),
t = "BULL", s* ∈ S

(Q11) Answer(Draw f)←
Sector(Icon f, Shares S), MAP(S, Title) = T,
T ⊃ {"BULL", "ICL", "SIEMENS"} , x = Draw(f)

(Q12) Answer(Name n, Set U)←
Share(Share* s*, Title t, Price p), Sector(Shares S, Name n)
p< 1000, s* ∈ S, U = GROUP(n,t), V = TOSET(U)

(Q13) Answer(Dtype i, Reduce r, List L)←
Dividend(Ddate d, Dtype i, Dividend m), d > 10-01-91, D = GROUP(i,d), L = TOLIST(D)
M = GROUP(i,m), N = TOSET(M), r = REDUCE(N,+,0)

(Q14) Answer(Flatten u, Name n)←
Sector(Name n, Shares S), T = Title.MAP(S), u = FLATTEN(t)

(V1) Closure(From p*, To q*)←
Connection(From P*, To q*), Part(Part* p*), Part (Part* q*)
Closure(From p*, To r*)←
Connection(From P*, To q*), Closure(From q*, To r*),
Part(Part* p*), Part (Part* q*), Part(Part* r*)

9.3 Open Issues

Although ESQL2 queries have a clear semantics as sketched above, further research is required to clarify certain semantic problems. We introduce a few topics that require further investigation below.

Infinite Queries

The semantics of an ESQL2 query is defined using extended DATALOG rules, which may be recursive. As the extended DATALOG includes methods, it is easy to generate queries with infinite answer. For example, the recursive rules:

P(int x) ← x =1
P(int x) ← P(int y) , x= y+1

defines a potentially infinite view.
Thus, the query :

SELECT * FROM P WHERE Int > 10

has an infinite answer, while the query :

SELECT * FROM P WHERE Int < 10

has a finite one.

Semantics of updates

ESQL2 supports updates. DATALOG in its basic form is monotonous and does not clearly support negations in rule heads. Further work in this area is required to model ESQL2 updates in a comprehensive DATALOG with updates.

Object invention

DATALOG rules do not make possible the creation of new objects, derived from existing objects or values. This feature, which has been introduced in DATALOG extensions [Abiteboul89] is not currently supported in ESQL2. An extension to save the result of a query as a new object is possible. The relationships of such an extension with the ESQL2 insert statements make it probably useless.

Collection support

DATALOG extended with collections should support collection unification. This has been done with sets, but not with more general collections. Sets, bags, lists and vectors are subtypes of collections. Thus, a collection is a generic type, which might be viewed as a union of its subtypes. To cope with duplicates in bags and orders in lists or vectors, it is possible to model a collection as a set of tuples, each tuple containing the number of occurrences and the value of an element. Thus, a collection could be modelled in DATALOG with sets and functions using a set of triples (v : element value, o : occurrence number, r : rank). This makes the modelling of collections rather complex.

10. Conclusion

In this paper, we have proposed a compatible extension of SQL, called ESQL2, which is a superset of ESQL2. The main extensions are in three directions: the support of complex data types whose instances are values, the support of classes whose instances are objects and the support of recursive rules as extended views. This demonstrates the integration in a common abstract data type framework of the relational, object-oriented and deductive paradigms. The integrated approach is based on the introduction of nested generic abstract data types and object identifiers. Compared to previous proposals [Beech88, Carey88,, Cluet90, Gardarin89a, Kim91, Scheck88, Valduriez89a], ESQL2 goes further in terms of relation and class support, query representation, nested data types, object-oriented and deductive capabilities. Furthermore, the integration of object-oriented features is based on a generic ADT framework, which is easy to implement inside a SQL system. ESQL2 also integrates uniformly the integrity facilities of SQL2. Two types of constraints are clearly identified: type constraints and table constraints.

The implementation of ESQL2 is going on in the EDS ESPRIT project in Europe. Classes are implemented as relations with a system attribute containing the object identifiers. Two ESQL2 prototypes have been demonstrated in ESPRIT weeks (1990 and 1991): one is the EDS distributed-memory parallel database server and the other the DBS3 shared-memory parallel database server [Bergsten 91]. In both implementations, the same compiler (with different cost models) is used. Two applications have been used as demonstration-vehicle: a portfolio management application and a geographical application. ESQL2 seems to be rather convenient for such applications.

Extensions are already planned for ESQL2. These include disjunctive types and multiple inheritance for types. Another planned extension is to support triggers to manage active databases. Concrete views are also under consideration. Interestingly, ESQL2 complies with all three tenets of third-generation DBMS as defined in [Beech90]. ESQL2 can therefore make a substantial contribution in the area of third-generation database-oriented programming languages.

References

[Abiteboul89] S. Abiteboul, P. Kanellakis, "Object Identity as a Query Language Primitive", ACM SIGMOD Int. Conf., Portland, Oregon, June 1989.

[Abiteboul90] S. Abiteboul, "Towards a Deductive Object-Oriented Database Language", Data & Knowledge Engineering, Vol. 5, No. 4, Oct. 1990.

[Bancilhon88] F. Bancilhon, "Object-Oriented Database Systems", Int. Symp. on PODS, Austin, Texas, March 1988.

[Beech88] D. Beech, "A Foundation for Evolution from Relational to Object Databases", Int. Conf. on EDBT, Venice, Italy, March 1988.

[Beech90] D. Beech, P. Bernstein, M. Brodie, M. Carey, J. Gray B. Lindsay, A. Rowe, M. Stonebraker, "Third-Generation database System Manifesto", Technical Report UCB/ERL M90/28, University of California, Berkeley, April 1990.

[Bergsten91] B. Bergsten, M. Couprie, P. Valduriez, "Prototyping DBS3, a Shared-Memory Parallel DBS", Int. Conf. on PDIS, Miami, Dec. 1991.

[Carey88] M.J. Carey, D.J. DeWitt, S.L. Vandenberg, "A Data Model and Query Language for EXODUS", ACM SIGMOD Int. Conf., Chicago, Illinois, June 1988.

[Cattell91] R.G. Cattell, "The Engineering Database Benchmark", in [Gray91].

[Danforth92] S. Danforth, P. Valduriez, "A Fad for Data-Intensive Applications", IEEE Trans. on Data and Knowledge Engineering, Vol. 3 , No. 2, February 92.

[Cluet90] S. Cluet, C. Delobel, C. Lecluse, P. Richard, "RELOOP, an algebra based Query Language for an Object-Oriented Database System", Data & Knowledge Engineering, Vol. 5, No. 4, Oct. 1990.

[Gardarin89a] G. Gardarin et al., "Managing Complex Objects in an Extended Relational DBMS", Int. Conf. on VLDB, Amsterdam, August 1989.

[Gardarin89b] G. Gardarin, P. Valduriez, "Relational Databases and Knowledge Bases", Book, 412 pages, Addison Wesley, New-York, 1989.

[Gardarin92a] G. Gardarin, P. Valduriez, "ESQL: An Object-Oriented SQL with F-Logic Semantics", Int. Conf. on Data Engineering, Phoenix, February 1992.

[Gardarin92b] G. Gardarin et. al., "ESQL: An Extended SQL with Object-Oriented and Deductive Capabilities", EDS Report EDS.DD.11B.4301, issue 3, available through INRIA, Paris, February 1992.

[Gray91] J. Gray Ed., "The Benchmark Handbook", Book, Morgan & Kaufman Pub., San Mateo, 1991.

[Guttag77] J. Guttag, "Abstract Data Types and the Development of Data Structures", Comm. of ACM, Vol. 20, No. 6, June 1977.

[ISO89] ISO/IEC, 2nd Edition, "Database Language SQL with Integrity Enhancement", ISO/DIS 9075, International Standard SQL1, 1986.

[ISO91] ISO/IEC JTC1/SC21 N5739, "Database Language SQL", ISO/DIS 9075:199x (E), Draft International Standard, April 1991.

[Khoshafian86] S. Khoshafian, G. Copeland, "Object Identity", Int. Conf. on OOPSLA, Portland, Oregon, September 1986.

[Khoshafian87] S. Khoshafian, P. Valduriez, "Persistence, Sharing and Object Orientation: a database perspective", Int. Workshop on Database Programming Languages, Roscoff, France, September 1987.

[Kim90] Won Kim, "Introduction to Object-Oriented Databases", Book, 234 pages, The MIT Press, Cambridge, Mass., 1990.

[Kifer89] M. Kifer, G. Lausen, "F-Logic: A Higher-Order Language for Reasoning about Object Identity, Inheritance and Schema", ACM SIGMOD Int. Conf., Portland, Oregon, June 1989.

[Ozsoyoglu87] Z.M. Ozsoyoglu, L-Y. Yuan, "A New Normal Form for Nested Relations", ACM TODS, Vol. 12, No. 1, March 1987.

[Schek88] H. Schek, "Nested Relations, a Step Forward or Backward", IEEE Bulletin on Data Engineering, Vol. 11, No. 3, September 1988.

[Shipman81]D. W. SHIPMAN, "The Functional Data Model and the Data Language Daplex", ACM TODS, March 1981, Vol. 6, No. 1.

[Shaw90] Shaw P., "Database Language Standards: Past, Present and Future", Lecture Notes in Computer Science, N. 466, Databases of the 90s, Springer-Verlag Ed., November 1990.

[Stonebraker83] M. Stonebraker, B. Rubenstein, A. Guttman, "Application of Abstract Data Types and Abstract Indices to CAD Databases", ACM SIGMOD Int. Conf., San Jose (California), May 1983.

[Stonebraker86] M. Stonebraker, L.A. Rowe, "The Design of POSTGRES", ACM SIGMOD Int. Conf., Washington, D.C., May 1986.

[Ullman88] J. D. Ullman, "Principles of Database and Knowledge-Base Systems", Book, Vol. 1, 1998, Computer Science Press.

[Valduriez89a] P. Valduriez, S. Danforth, "Functional SQL (FSQL), an SQL Upward Compatible Database Programming Language", MCC Technical Report ACA-ST-045-89, February 1989, to appear in Information Sciences.

[Valduriez89b] P. Valduriez, S. Danforth, B. Hart, T. Briggs, M. Cochinwala, "Compiling FAD, a Database Programming Language", Int. Workshop on Database Programming Languages, Portland, Oregon, June 1989.

ANNEX
Syntax of ESQL2

This annex defines the syntax of the main ESQL2 commands. It is built from a subset of the ISO Data Base Languages reference document of May 1991 (ISO/TEC JTC1 WG3 DBL CAN). When an element is not defined in the following syntax, it is a standard SQL2 element as defined by ISO. Thus, the reader should refer to the ISO document to understand the syntax of the undefined elements. As much as possible, we try to give a comprehensible set of definitions without too much implicit references to the ISO document. We detail extensions and refer only to the generally well known part of SQL.

1. Schema Definition

Function :

Define a database schema.

Format :

<schema definition> ::=
CREATE SCHEMA <schema authorization clause>
[<schema element> ...]

<schema authorization clause> ::=
<schema name>
| AUTHORIZATION <user authorization identifier>
| <schema name> AUTHORIZATION <user authorization identifier>

<schema element> ::=
<data type definition>
| <domain definition>
| <entity definition>
| <view definition>
| <grant statement>
| <assertion definition>
| <trigger definition>

General rules :

A database is the collection of all data defined in a schema. A schema is either identified by a name, or a user authorization identifier or both.

2. Entity Definition

Function :

Define a persistent base table or class, which is composed of simple or complex values or references, as specified by the column data type. The base table or class can be defined from more general tables or classes.

Format :

<entity definition> ::=
CREATE {TABLE | CLASSE} <entity name>
{<entity elements> | <subentity clause> [<entity elements>]}

<entity name> ::=
<table name>
| <class name>

<entity elements> ::=
(<entity element> [{,<entity element>}...])
<entity element> ::=
<column definition>
| <entity constraint definition>

<column definition> ::=
<column name> <data type> [<type constraint definition>]

<subentity clause> ::=
{SUBTABLE OF | ISA} <genentity> [{, <genentity>}...]
[AS (<column name>[{,<column name>}...])]
<genentity> ::= <entity name>

General rules :

The table definition of SQL2 is extended to support classes. Note that certain SQL3 proposals support subtables, which have been included in ESQL2 as subentities. If ISA is specified, then the created entity inherits all the columns of the generic entities (<genentity>) in the given order, except the identifier when a table inherits from a class. If AS is specified, then the <column name>s of inherited columns are replaced by the corresponding <column name>s specified following AS. A not renamed column is inherited only once if it appears in several generic tables with the same name; the first occurrence of the column is then inherited.

3. Entity Alteration

Function :

Change a table or class and its definition by adding specific columns or removing existing ones, or by adding new constraints or removing existing ones.

Format :

<alter table statement ::=
ALTER {TABLE | CLASS} <entity name> <alter action>
<alter action> ::=
<add column definition>
| <drop column definition>
| <add entity constraint definition>
| <drop entity constraint definition>

<add column definition> ::=
ADD <column definition>
<drop column definition> ::=

DROP <column name> [CASCADE]

<add entity constraint definition> ::=

ADD <entity constraint definition>

<drop entity constraint definition> ::=

DROP CONSTRAINT <constraint name> [CASCADE]

General rules :

If the ADD clause is specified, the new column name must not already exist within the given entity. If the DROP clause is specified, the given column must already exists within the given entity. If CASCADE is specified, the column is suppressed from all the subentities. Similar rules hold for integrity constraints.

4. User Data Type Definition

Function :

Define a user data type with its structure and specify whether type instances are values or references to objects within a class.

Format :

<type definition> ::=

CREATE TYPE <type name> [AS] <complex data type>
[WITH FUNCTION <function specification> [{,<function specification>}...]]
[<type constraint definition> [{,<type constraint definition>}...]]

<complex data type> ::=

{SUBTYPE OF | ISA} <type name>
[WITH (<column name> <type clause>
[{ ,<column name> <type clause> } ...])]
[OMIT (<function name> [{,<function name>}...])]
| <type clause>

<type clause> ::=

<structure clause>
| < data type >

<structure clause> ::=

[TUPLE [OF]] (<column name> <type clause> [{ ,<column name> <type clause> } ...])
| <generic type> [OF] <type clause> [{, <type clause> }...]

<generic type> ::= SET | BAG | LIST | VECTOR | <constructor>

<data type> ::=

<character string type>
| <national character string type>
| <numeric type>
| <enumerated type>
| <datetime type>
| <type name>
| <class name>

General rules :

If SUBTYPE or ISA is specified, the user data type is a specialization of one or more previously defined types; then the created types inherits all the properties of the given super-type (<type name>) in the given order. If WITH is specified, then the column names

that follows WITH are added to the super-type, which must be a tuple type. If certain column names already exist in the super-type, they are replaced by the corresponding column names specified following WITH.

The list of generic types (also called constructors) corresponds to the library of generic types supplied with the system. With each constructor comes a library of defined functions.

If a class name is used as a data type, the data type is an object identifier referencing an object in the class. Note that the structure clause can itself include object identifiers. Considering a class as defining a reference data type allows ESQL to support referential sharing.

5. User Function Defintion

Function :

Create a new function for an existing data type.

Format :

CREATE FUNCTION <function specification>
[APPLY [ON] <type name>]

<function specification> ::=
<function name>(<operand data type list>)
[**RETURNS** <result data type>]
<language clause>
<file clause>

<operand data type list> ::= <data type> [{,data type}...]
<result data type> ::= data type

<language clause> ::= **LANGUAGE** { C | C++ | LISP | PROLOG | <language name> }

<file clause ::= **FILE** <file name>

General rules :

The operand data type list is a list of data types (simple or complex); the result data type is simply the data type of the function answer. The language clause specifies in which language the ADT function is programmed. The file clause specifies in which file is located the text of the function.

6. User Data Type Alteration

Function :

Change a type and its definition by adding specific functions or removing them, or by adding new columns or removing existing ones (only valid for a tuple type).

Format :

<alter type statement> ::=
ALTER TYPE <type name> <alter type list>
<alter type list> ::=
<alter type element> [{,<alter type element>}...]
<alter type element> ::=

<add function clause> | <drop function clause>
| <add column clause> | <drop column clause>

<add function clause> ::=
ADD FUNCTION <function specification>
<drop function clause> ::=
DROP FUNCTION <function name>
<add column clause> ::=
ADD [COLUMN] <column name> <type clause>
<drop column clause> ::=
DROP [COLUMN] <column name>

General rules :

If the ADD FUNCTION clause is specified, a new function is defined to encapsulate the altered user data type. Thus, this user data type must appear as the first argument of the function. Note that the syntax of the ADDFUN clause is similar to that of the DECLARE EXTERNAL procedure of SQL3.

If the DROP FUNCTION clause is specified, the given function name must already exists within the given user type.

If the ADD COLUMN clause is specified, the user type must be constructed as a tuple : a new column is then added to the tuple. If the DROP COLUMN clause is specified, the user type must be a tuple with the given column.

7. User Data Type Deletion

Function :

Suppress a user data type and its definition.

Format :

<drop type statement> ::=
DROP TYPE <type name>

General rules :

The type name must have been defined before.

8. Query Specification

Function :

Specify a table of simple or complex objects derived from the result of a query expression, composed of union, difference (EXCEPT) and intersection of SELECT statements.

Format :

<query expression> ::=
<query term>
| <query expression> **[OUTER] UNION [ALL]** <query term>
| <query expression> **EXCEPT [ALL]** <query term>

<query term> ::=

```
        <query specification>
|       <query term> INTERSECT [ALL] <query specification>

<query specification> ::=
        SELECT [ALL | DISTINCT] <select list> <table expression>

<select list> ::=
                *
        |       <select sublist> [ { , <select sublist> } ...]

<select sublist> ::= <derived column> | <qualifier>.*

<derived column> ::= <value expression> [ AS <column name> ]

<table expression> ::=
        <from clause>
        [<where clause>]
        [<group by clause>]
        [<having clause>]
        [<order by clause>]

<group by clause> ::=
        GROUP BY <column specification> [{,<column specification>}...]

<value expression> ::=
                <numeric value expression>
        |       <character value expression>
        |       <datetime value expression>
        |       <interval value expression>
        |       <complex data type value expression>

<complex data type value expression> ::=
                <complex data type primary>
        |       <function> ( <operand> [ { , <operand>} ...] )

<operand> ::=   < value expression>

<complex data type primary> ::=
                <column specification>
        |       <class name>*
        |       <complex data type primary>.<function> ( <operand> [ { , <operand>} ...] )
```

General rules :

Queries are similar to SQL2 queries with extensions for handling methods. A query expression is a union of intersection of simple queries, with possible differences. If ALL is not specified, then UNION, INTERSECTION and EXCEPT are understood without duplicates. If ALL is specified, the number of duplicates in the tables is considered; more precisely, let T1 and T2 be two tables having respectively t1 and t2 occurrences of a tuple t. Then, the number of occurrences of t of T1 UNION ALL T2 is t1 + t2; the number of t's occurrences of T1 EXCEPT ALL T2 is the maximum of (t1 - t2) and zero; the number of t's occurrences of T1 UNION ALL T2 is the minimum of t1 and t2.

A corresponding specification is not permitted as in SQL2. Thus, the two tables which appear as operators of a set operation (UNION, INTERSECT, EXCEPT) must be union compatible. That condition can be achieved by renaming the attributes of the SELECT list in one of the table (see SELECT for syntax) if necessary.

The extensions for handling methods concern the definition of the <value expression>, which may now involve function calls. In general, functions are applied to a complex data type primary using the dot notation, which is the first function argument. Certain functions (e.g., the MAKE functions) with an empty first argument can be applied directly to a list of operands. Note that a complex data type expression may be used in a SELECT list expression, in a WHERE condition expression, in a HAVING condition expression and in an ORDER BY sort specification. The only requirement is that the function return a correct data type for the expression or the predicate; this data type may be complex. Note also that a library of built in functions comes with the generic data types (constructors).

9. Generalized View Creation

Function :

Create a generalized view, which may possibly be recursive. A check option constraints can be defined for an updatable view. A rule is defined as a query specification which may refer to the view in the FROM clause (recursive views).

Format :

<deductive view definition> ::=
CREATE VIEW <table name> [(<view colum list>)]
AS <query expression>
[RECURS <numeric value expression> **TIMES]**
[**WITH CHECK OPTION** <constraint name definition>]

<view column list> ::=
<column name> [{ , <column name> } ...]

General rules :

The query expression can refer to the view table name (recursive views). In that case, the RECURS clause may be specified. It limits the level of recursion (i.e., the number of times the view is searched) to the entire value of the numeric value expression. Only simple views, which are not recursive and do not use set operation, are updatable and can support a check option constraint (WITH CHECK OPTION).

10. Generalized View Deletion

Function :

Drop a view.

Format :

<deductive view deletion> ::=
DROP VIEW <table name>

General rule :

The table name must have been defined as a view.

11. Update Specification

Function :

Update rows of a table, including referenced objects.

Format :

<searched update statement > ::=
UPDATE <entity name>
SET <searched set clause > [{ , <searched set clause> } ...]
[WHERE <search condition>]

<searched set clause> ::=
<assignation set clause> | <application set clause>

<assignation set clause> ::=
<column name> = { <value expression> | <null value> }

<application set clause> ::=
<column name>.<function> [(<operand> [{ , <operand>} ...])]

General rules :

If the column name is of a standard SQL2 type, then the value expression must be of similar type and the column value is simply assigned to the <value expression>. In that case, the assignation set clause must be used.

If the column name is a complex data type value expression, the two variants are possible. The assignation set clause is used when the value expression computes a value of similar type of the column name, to perform a full update of the column. The application set clause is used to apply an update function to the column name. Operands are possible, according to the function definition.

12. Insertion Specification

Function :

Insert rows in a table, including referenced objects.

Format :

<insert statement> ::=
INSERT INTO {<entity name>} [(<insert column list>)]
<query specification>
| **VALUES** <row value expression> [{ , <row value expression > } ...]

<insert column list> ::=
<column name> [{ , <column name > } ...]

<row value expression> ::=
<value expression>
| <null value>
| (<value expression list>)

<value expression list> ::= <list element> [{ , <list element> } ...]

<list element> ::=
<value expression>
| <null value>

<null value> ::= NULL

General rules :

The command is basically not modified from standard SQL2. Note however that a value expression can include complex objects or references to complex objects, with method invocations.

13. Deletion Specification

Function :

Delete rows in a table, including referenced objects if they are no more referenced.

Format :

<delete statement> ::=
DELETE FROM <entity name>
[**WHERE** <search condition>]

General rules :

The command is basically not modified from standard SQL2. Note however that a search condition can involve complex objects and method applications.

14. Integrity Constraint Definition

Function :

Define data type and table integrity constraints, including domains, unique key and referential constraints.

Format :

<integrity constraint definition> ::=
CREATE CONSTRAINT <constraint name>
{ <type constraint definition> | <entity constraint definition> }

<type constraint definition> ::=
[[**ON**] **TYPE** <data type>]
CHECK (<search condition>)

<table constraint definition> ::=
[[**ON**] {**TABLE** | **CLASS**} <entity name>]
{ <unique constraint definition>
| <referential constraint definition>
| <check constraint definition> }

<unique constraint definition> ::=
{ **UNIQUE** | **PRIMARY KEY** } (<unique column list>)
<unique column list> ::= <column name> [{,<column name>}...]

<referential constraint definition> ::=
<referential constraint> [<referential triggered action>]
<referential constraint> ::=
FOREIGN KEY (<referencing column list>)
<references specification>
<referencing column list> ::= <column name> [{,<column name>)...]
<references specification> ::=
REFERENCES <entity name> [(<referenced column list>)]
<referenced column list> ::= <column name> [{,<column name>)...]
<referential triggered action> ::=
<update rule> [<delete rule>]
| <delete rule> [<update rule>]
<update rule> ::= **ON UPDATE** { **CASCADE** | **SET NULL**}
<delete rule> ::= **ON DELETE** { **CASCADE** | **SET NULL**}

<check constraint definition> ::=
CHECK (<search condition>)

General rules :

In general, constraint definition and verification are similar to the SQL2 standard, except that constraints are extended to complex types. Type constraint and entity constraint are effectively checked after the execution of each SQL statement which may violate them.

The referencing entity and the referenced entity satisfy the referential constraint definition if and only if for each referencing column list value in the refencing table either the value is null or the value exists in the referenced table as a value of the referenced column list. If no column is specified in the referenced entity, the primary is assumed for a table and the object identifier is assumed for a class.

15. Integrity Constraint Deletion

Function :

Drop an integrity constraint.

Format :

<integrity constraint deletion> ::=
DROP CONSTRAINT <constraint name>

General rule :

The constraint must have been created before under the given name.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique

ISSN 0249-6399